Transactional Memory

Part 1: Concepts and Hardware-Based Approaches
Introduction

- Provide support for concurrent activity using transaction-style semantics without explicit locking
- Avoids problems with explicit locking
  - Software engineering problems
  - Priority inversion
  - Convoying
  - Deadlock
- Approaches
  - Hardware (faster, size-limitations, platform dependent)
  - Software (slower, unlimited size, platform independent)
    - Word-based (fine-grain, complex data structures)
    - Object-based (course-grain, higher-level structures)
History

Lomet* proposed the construct:

```
<identifier> : action( <parameter-list> ) ;
<statement-list>
end;
```

where the statement-list is executed as an atomic action. The statement-list can include:

```
await <test> then <statement-list> ;
```

so that execution of the process/thread does not proceed until test is true.

Transaction Pattern

repeat {

    BeginTransaction(); /* initialize transaction */
    <read input values>
    success = Validate(); /* test if inputs consistent */
    if (success) {
        <generate updates>
        success = Commit(); /* attempt permanent update */
        if (!success)
            Abort(); /* terminate if unable to commit */
    }
    EndTransaction(); /* close transaction */

} until (success);
Guarantees

- **Wait-freedom**
  - All processes make progress in a finite number of their individual steps
  - Avoid deadlocks and starvation
  - Strongest guarantee but difficult to provide in practice

- **Lock-freedom**
  - At least one process makes progress in a finite number of steps
  - Avoids deadlock but not starvation

- **Obstruction-freedom**
  - At least one process makes progress in a finite number of its own steps in the absence of contention
  - Avoids deadlock but not livelock
  - Livelock controlled by:
    - Exponential back-off
    - Contention management
Hardware Instructions

Compare-and-Swap (CAS):

```c
word CAS (word* addr, word test, word new) {
    atomic {
        if (*addr == test) {
            *addr = new;
            return test;
        }
        else return *addr;
    }
}
```

Usage: a spin-lock

```c
inuse = false;
...
while (CAS(&inuse, false, true);
```

Examples: CMPXCHNG instruction on the x86 and Itaninimum architectures
Hardware Instructions

LL/SC: load-linked/store-conditional

```c
word LL(word* address) {
    return *address;
}

boolean SC(word* address, word value){
    atomic {
        if (address updated since LL)
            return false;
        else {
            address = value;
            return true;
        }
    }
}
```

Usage:
```
repeat {
    while (LL(inuse));
    done = SC(inuse, 1);
} until (done);
```

Examples: ldl_l/stl_c and ldq_l/stq_c (Alpha), lwarx/stwcx (PowerPC), ll/sc (MIPS), and ldrex/strex (ARM version 6 and above).
Hardware-based Approach

- Replace short critical sections
- Instructions
  - **Memory**
    - Load-transactional (LT)
    - Load-transactional-exclusive (LTX)
    - Store-transactional (ST)
  - **Transaction state**
    - Commit
    - Abort
    - Validate
- Usage pattern
  - Use LT or LTX to read from a set of locations
  - Use Validate to ensure consistency of read values
  - Use ST to update memory locations
  - Use Commit to make changes permanent
- Definitions
  - Read set: locations read by LT
  - Write set: locations accessed by LTX or ST
  - Data set: union of Read set and Write set
Example

typedef struct list_elem { struct list_elem *next; /* next to dequeue */
    struct list_elem *prev; /* previously enqueued */
    int value; } entry;

shared entry *Head, *Tail;
void list_enq(entry* new) {
    entry *old_tail;
    unsigned backoff = BACKOFF_MIN;
    unsigned wait;

    new->next = new->prev = NULL;
    while (TRUE) {
        old_tail = (entry*) &Tail;
        if (VALIDATE()) {
            ST(&new->prev, old_tail);
            if (old_tail == NULL) { ST(&Head, new); } else { ST(&old_tail->next, new); } ST(&Tail, new);
            if (COMMIT) return;
        }
        wait = random() % (01 << backoff); /* exponential backoff */
        while (wait--);
        if (backoff < BACKOFF_MAX) backoff++;
    }
}
Hardware-based Approach
Cache Implementation

- Processor caches and shared memory connected via shared bus.
- Caches and shared memory “snoop” on the bus and react (by updating their contents) based on observed bus traffic.
- Each cache contains an (address, value) pair and a state; transactional memory adds a tag.
- Cache coherence: the (address, value) pairs must be consistent across the set of caches.
- Basic idea: “any protocol capable of detecting accessibility conflicts can also detect transaction conflict at no extra cost.”
## Line States

<table>
<thead>
<tr>
<th>Name</th>
<th>Access</th>
<th>Shared?</th>
<th>Modified?</th>
</tr>
</thead>
<tbody>
<tr>
<td>invalid</td>
<td>none</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>valid</td>
<td>R</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>dirty</td>
<td>R, W</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>reserved</td>
<td>R, W</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

### Diagram

- **Cache**
- **Shared Memory**
- **Bus**
- **address**
- **value**
- **state**
- **tags**
Transactional Tags

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMPTY</td>
<td>contains no data</td>
</tr>
<tr>
<td>NORMAL</td>
<td>contains committed data</td>
</tr>
<tr>
<td>XCOMMIT</td>
<td>discard on commit</td>
</tr>
<tr>
<td>XABORT</td>
<td>discard on abort</td>
</tr>
</tbody>
</table>

Shared Memory

Bus

Cache

<table>
<thead>
<tr>
<th>address</th>
<th>value</th>
<th>state</th>
<th>tags</th>
</tr>
</thead>
</table>
**Bus cycles**

<table>
<thead>
<tr>
<th>Name</th>
<th>Kind</th>
<th>Meaning</th>
<th>New access</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ</td>
<td>regular</td>
<td>read value</td>
<td>shared</td>
</tr>
<tr>
<td>RFO</td>
<td>regular</td>
<td>read value</td>
<td>exclusive</td>
</tr>
<tr>
<td>WRITE</td>
<td>both</td>
<td>write back</td>
<td>exclusive</td>
</tr>
<tr>
<td>T_READ</td>
<td>transaction</td>
<td>read value</td>
<td>shared</td>
</tr>
<tr>
<td>T_WRITE</td>
<td>transaction</td>
<td>read value</td>
<td>exclusive</td>
</tr>
<tr>
<td>BUSY</td>
<td>transaction</td>
<td>refuse access</td>
<td>unchanged</td>
</tr>
</tbody>
</table>
Scenarios

- **LT instruction**
  - If XABORT entry in transactional cache: return value
  - If NORMAL entry
    - Change NORMAL to XABORT
    - Allocate second entry with XCOMMIT (same data)
    - Return value
  - Otherwise
    - Issue T_READ bus cycle
    - **Successful**: set up XABORT/XCOMMIT entries
    - **BUSY**: abort transaction

- **LTX instruction**
  - Same as LT instruction except that T_RFO bus cycle is used instead and cache line state is RESERVED

- **ST instruction**
  - Same as LTX except that the XABORT value is updated
Performance Simulations

comparison methods

- TTS – test/test-and-set  
  (to implement a spin lock)
- LL/SC – load-linked/store-conditional  
  (to implement a spin lock)
- MCS – software queueing
- QOSB – hardware queueing
- Transactional Memory